

The Strange Case of
Dr. Admissibility
and Mr. Derive

Dan Licata

Joint work with Noam Zeilberger and Robert Harper

Goal

A programming language that helps people:

- * Define programming languages and logics
- * Study their properties
- * Use logics to reason about programs

Need

A programming language that makes it easy to:

- * Represent syntax as data:
 - * Programs of a programming language
 - * Propositions of a logic
- * Compute with these representations:
 - * Write a compiler
 - * Write a theorem prover

Derivability

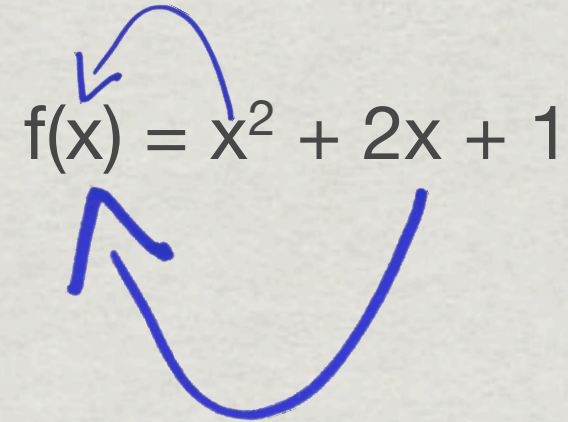
In math

Polynomials over the reals:

$$f(x) = x^2 + 2x + 1$$

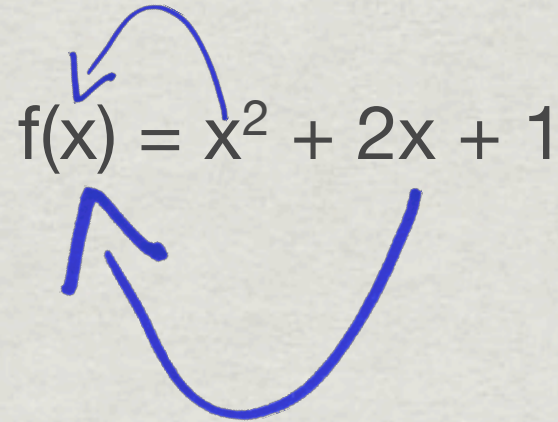
In math

Polynomials over the reals:

$$f(x) = x^2 + 2x + 1$$
The equation $f(x) = x^2 + 2x + 1$ is shown with blue hand-drawn annotations. A curved arrow points from the x^2 term to the $2x$ term, and another curved arrow points from the $2x$ term to the constant 1 . Below the equation, a blue parabola is drawn, opening upwards, with its vertex at the origin, representing the function $f(x) = (x+1)^2$.

In math

Polynomials over the reals:

$$f(x) = x^2 + 2x + 1$$


Substitution: plug in for the variable

* $f(3) = 3^2 + 2 \cdot 3 + 1$

* $f(y+5) = (y+5)^2 + 2(y+5) + 1$

In programming

Functional abstraction:

```
fact(x) = if (x = 0)
          then 1
          else x * (fact (x-1))
```


In programming

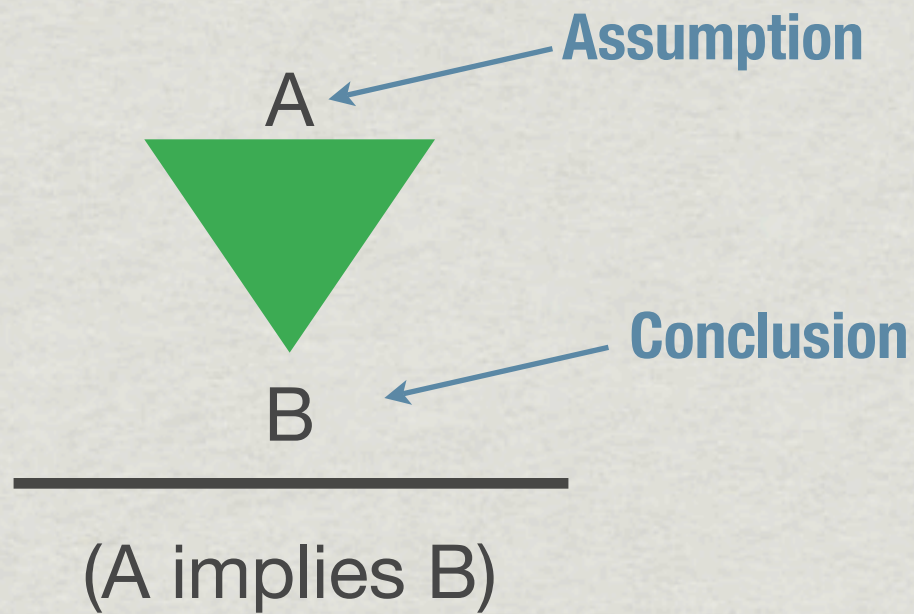
Functional abstraction:

```
fact(x) = if (x = 0)
          then 1
          else x * (fact (x-1))
```

Compute by substitution:

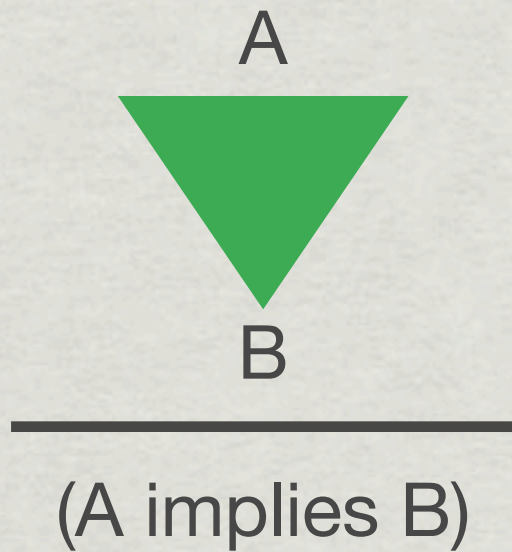
```
fact(5) = if (5 = 0)
          then 1
          else 5 * (fact (5-1))
```

In logic



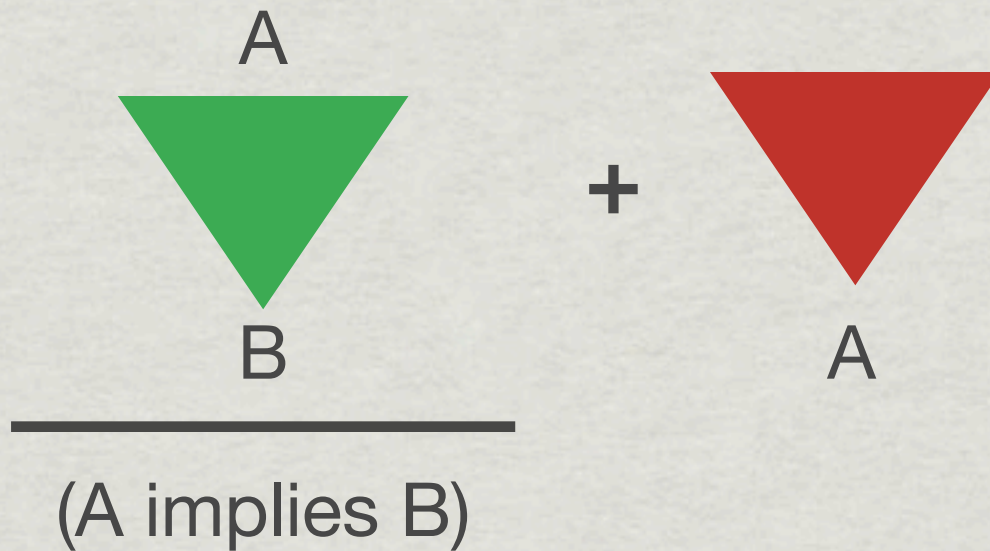
In logic

If (A implies B) and A then B



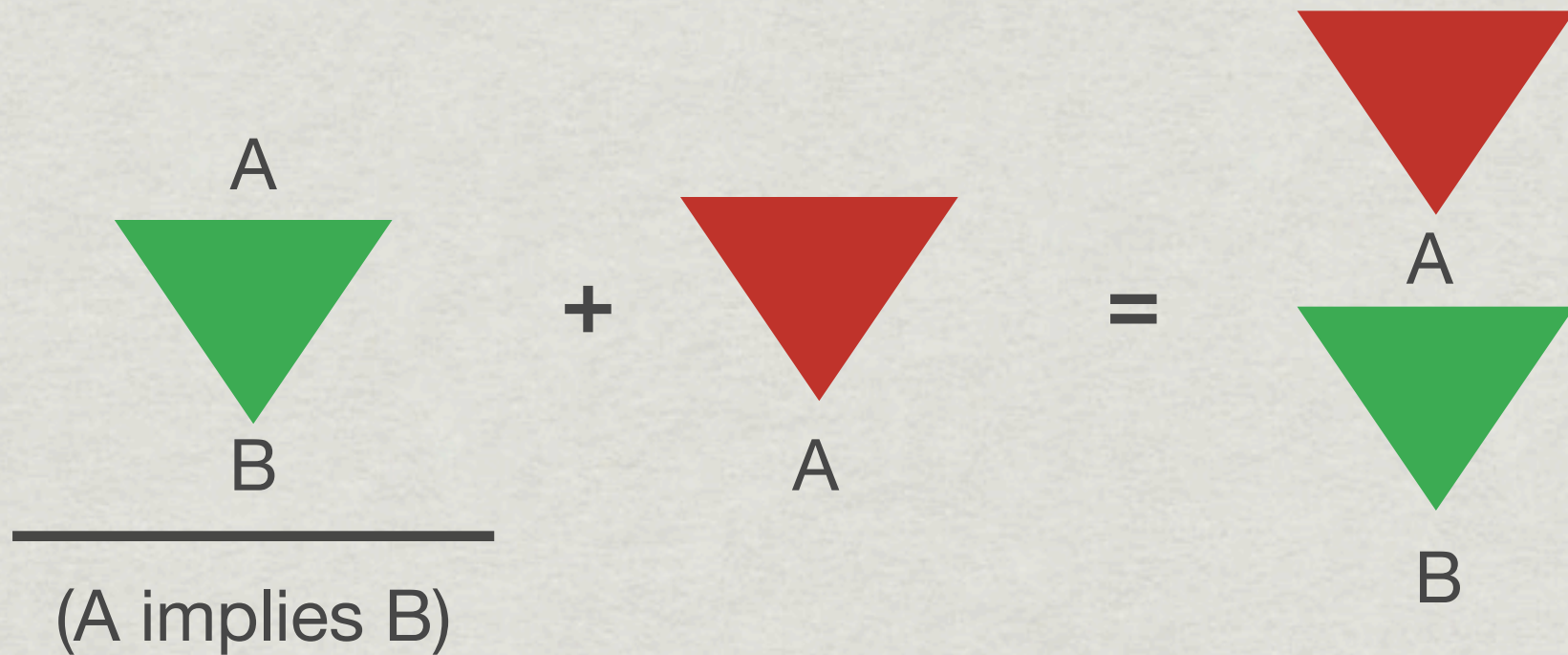
In logic

If (A implies B) and A then B



In logic

If (A implies B) and A then B



Derivability

*Captures notion of **terms-with-holes**,
which can be filled by substitution*

- * Represent syntax with variable binding
- * Represent hypothetical judgements (typing for a programming language)

Admissibility

In math

Function from reals to reals specified by:

- * set of ordered pairs
- * every number appears exactly once on the LHS

$$\left\{ \begin{array}{l} (0, 1), \\ (1, 4), \\ (\sqrt{2}, 3 + 2\sqrt{2}), \\ \dots \end{array} \right\}$$

In programming

Specify a function by its behavior (e.g., memo table):

Factorial = { (0 , 1),
(1 , 1),
(2 , 2),
(3 , 6),
(4 , 24),
... }

In logic

To prove $\forall x:\text{nat}.A(x)$,
prove $A(n)$ for each numeral n .

$A(0)$ and
 $A(1)$ and
 $A(2)$ and
 $A(3)$ and
 $A(4)$...

In logic

To prove $\forall x:\text{nat.}(\text{even}(x) \vee \text{odd}(x))$,
prove $(\text{even}(n) \vee \text{odd}(n))$ for each numeral n .

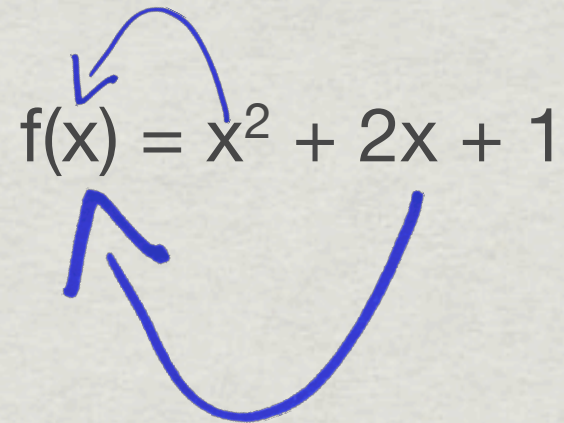
$\text{even}(0) \vee \text{odd}(0)$ and
 $\text{even}(1) \vee \text{odd}(1)$ and
 $\text{even}(2) \vee \text{odd}(2)$ and
 $\text{even}(3) \vee \text{odd}(3) \dots$

Admissibility

*Captures notion of an **arbitrary transformation**:
only i/o-behavior matters*

- * Represent proofs with infinitely many cases
- * Compute with logical systems (compiler transformation, translation between logics)
- * Prove theorems *about* logics

Derivability

$$f(x) = x^2 + 2x + 1$$


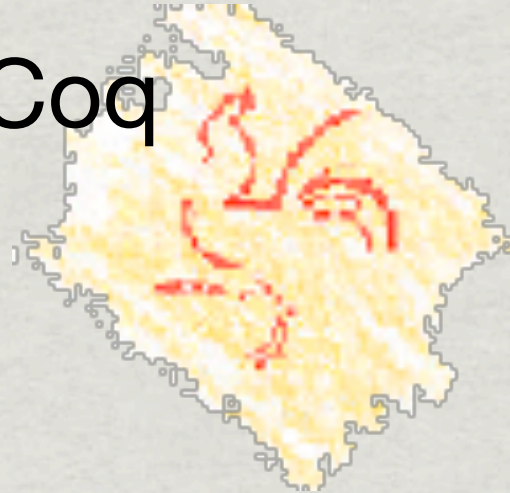
Admissibility

$$\{ (0, 1), \\ (1, 4), \\ (\sqrt{2}, 3 + 2\sqrt{2}), \\ \dots \}$$

Proof assistants



Coq



How well do existing proof assistants support derivability and admissibility?

Contributions

Definitions using

- * **Derivability**: easy in **Twelf**, hard in **Coq**
- * **Admissibility**: easy in **Coq**, hard in **Twelf**

Contributions

Definitions using

- * **Derivability**: easy in **Twelf**, hard in **Coq**
- * **Admissibility**: easy in **Coq**, hard in **Twelf**

This work:

***Supports definitions using
both derivability and admissibility***

Outline

- * Representations using **derivability**
- * Representations using **admissibility**
- * Mixed representations

Outline

- * Representations using **derivability**
- * Representations using **admissibility**
- * Mixed representations

Derivability

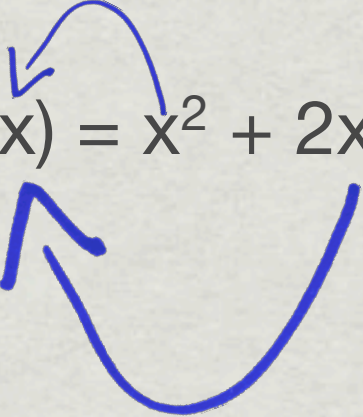
How do we represent **variable binding**?

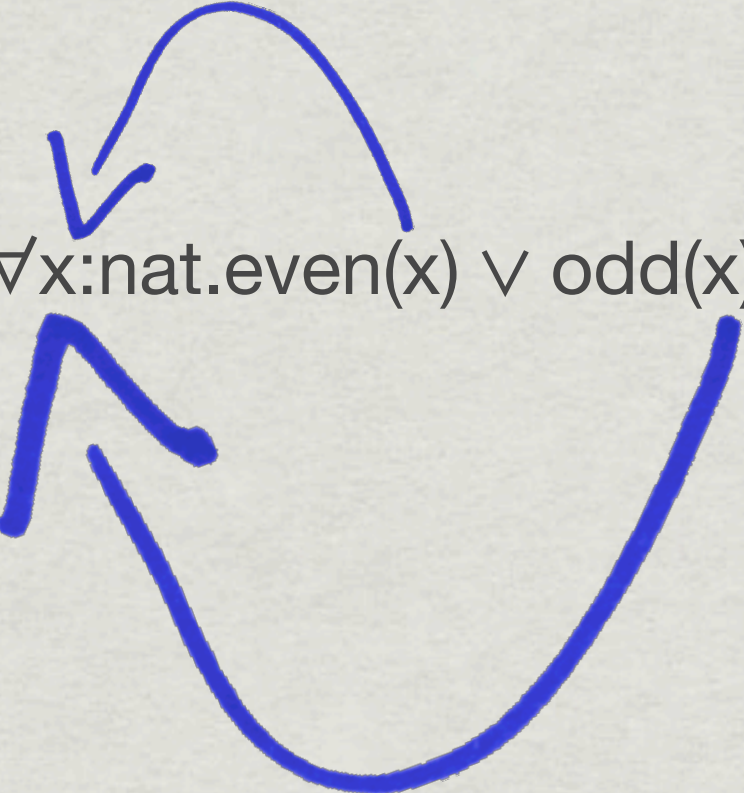
Derivability

How do we represent **variable binding**?

*What is **variable binding**?*

Syntax with binding

$$f(x) = x^2 + 2x + 1$$


$$\forall x:\text{nat}.\text{even}(x) \vee \text{odd}(x)$$


Structural properties

Properties of variables:

- * Substitution
- * Renaming
- * Weakening

Structural properties

Properties of variables:

- * Substitution (can plug in)
- * Renaming
- * Weakening

Structural properties

Properties of variables:

- * Substitution (can plug in)
- * **Renaming**
- * Weakening

Dan's Homework

1. Factor the following: $f(x) = x^2 + 2x + 1$

Answer :

$$f(y) = (y + 1)^2$$

Dan's Homework

1. Factor the following: $f(x) = x^2 + 2x + 1$

Answer :

$$\del f(y) = (y + 1)^2$$

The answer is
 $f(x) = (x + 1)^2$

Dan's Homework

1. Factor the following: $f(x) = x^2 + 2x + 1$

Answer :

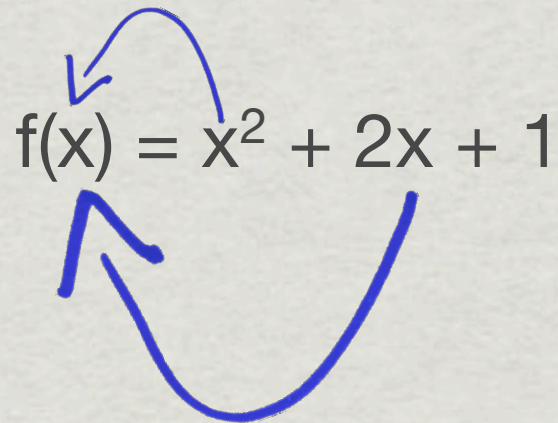
$$\del{f(y) = (y + 1)^2}$$

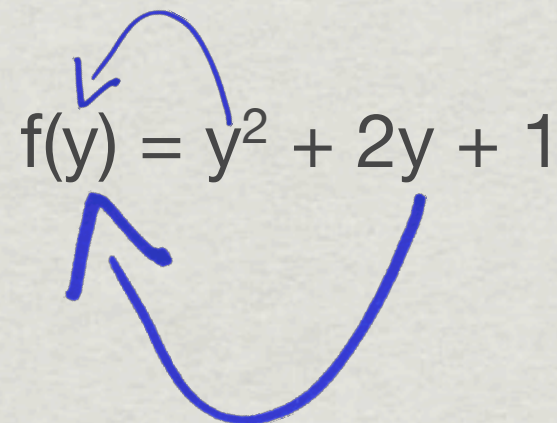
The answer is
 $f(x) = (x + 1)^2$

(Dan finds a new math teacher...)

Renaming

Variables are **pronouns**:
pointer structure matters, names don't

$$f(x) = x^2 + 2x + 1$$


$$f(y) = y^2 + 2y + 1$$


Renaming

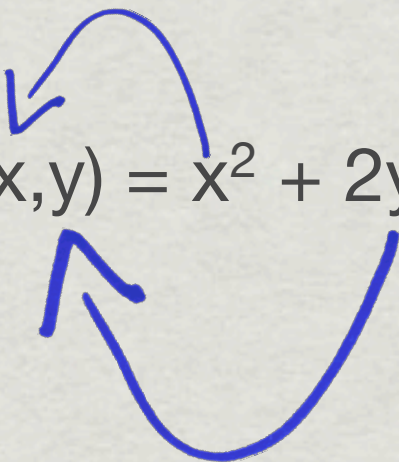
Variables are **pronouns**:
pointer structure matters, names don't

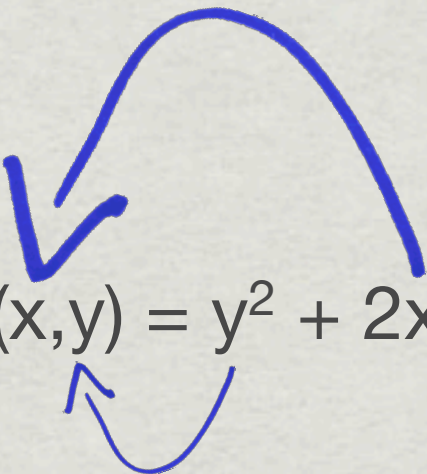
$$f(x,y) = x^2 + 2y + 1$$

$$f(y,x) = y^2 + 2x + 1$$

Renaming

Variables are **pronouns**:
pointer structure matters, names don't

$$f(x,y) = x^2 + 2y + 1$$


$$f(x,y) = y^2 + 2x + 1$$


Structural properties

Properties of variables:

- * Substitution (can plug in)
- * Renaming (names don't matter)
- * **Weakening**

Weakening

“A polynomial in **one variable** is called a **univariate polynomial**, a polynomial in **more than one variable** is called a **multivariate polynomial** ... when working with **univariate polynomials** one **does not exclude constant polynomials**...although strictly speaking **constant polynomials do not contain any variables** at all.”

-- “Polynomial” in Wikipedia

Weakening

OK not to use a variable:

$$f(x) = 4$$

$$f(x,y) = x^2 + 2x + 1$$

Structural properties

Properties of variables:

- * Substitution (can plug in)
- * Renaming (names don't matter)
- * Weakening (unused variables ok)

Representing derivability

Q: How to implement data with binding?

Answer 1: variables = strings

Polynomial: $f(x) = x^2 + 2 * x + 1$

Representation: ("x", "x"² + 2 * "x" + 1)

name of bound variable



reference to binding



Representing derivability

Q: How to implement data with binding?

Answer 1: variables = strings

(“x”, “x”² + 2 * “x” + 1)

Problems:

- * Non-unique representations
- * Do functions respect renaming?
- * Have to implement substitution for each language

Representing derivability

Q: How to implement data with binding?

Other implement:

- * de Bruijn indices/levels
- * nominal logic [Pitts and Gabbay]
- * cofinite quantification [Ayedemir et al.]

Representing derivability

Q: How to implement data with binding?

Other implement:

- * de Bruijn indices/levels
- * nominal logic [Pitts and Gabbay]
- * cofinite quantification [Ayedemir et al.]

*These are **implementation**; we want the **interface**!*

Representing derivability

Key idea: represent **binding** with functions [HHP'93]

- * Language provides a type of **substitution functions** with substitution, weakening, ...
- * Implementation hidden from programmer

Representing derivability

$$f(x) = x + 3$$

represented by

$\lambda x.plus x 3$

Substitution functions written $\lambda u.V$, where V is a **value** (e.g. **constructors** and variables)

Representing derivability

Constructors generate syntax:

* `const` : rational \Rightarrow exp

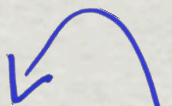
Example: 3 represented by `(const 3)`

* `plus` : exp \Rightarrow exp \Rightarrow exp

Example: 3 + 3 represented by
`plus (const 3) (const 3)`

Representing derivability


Polynomials represented by substitution functions of type $(\text{exp} \Rightarrow \text{exp})$:

$$f(x) = x + 3$$


**variables represented
by variables!**

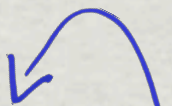
represented by

$\lambda x.$ plus x (const 3)



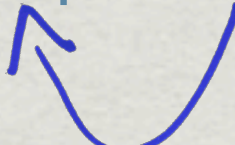
Representing derivability

Polynomials represented by substitution functions of type $(\text{exp} \Rightarrow \text{exp})$:

$$f(x) = x + 3$$


represented by

$\lambda x.\text{plus } x \text{ (const 3)}$



substitution = application


$f(5)$

represented by


$(\lambda x.\text{plus } x \text{ (const 3)})$
applied to $(\text{const } 5)$

Representing derivability

Polynomials represented by substitution functions of type $(\text{exp} \Rightarrow \text{exp})$:

$$f(x) = x + 3$$


represented by

$$\lambda x. \text{plus } x \text{ (const 3)}$$


substitution = application

$$f(5)$$

represented by

$$(\lambda x. \text{plus } x \text{ (const 3)})$$

applied to (const 5)

=

$$\text{plus (const 5) (const 3)}$$

Adequacy

- * Important that $\lambda u.V$ allows only substitution functions
- * Cannot use **admissibility** to represent syntax: there are many more **set-theoretic functions from expressions to expressions** than polynomials

$\{(\text{const } n, 2 * 2 * 2 * \dots * 2), \dots\}$


n times in total

Derivability

A theory of syntax with variable binding

1. Represent data with binding

a) Define constructors (**plus**)

b) Use derivability to represent binding (\Rightarrow)

2. Next: write recursive proofs/programs
(**admissibility**)

Outline

- * Representations using **derivability**
- * **Representations using admissibility**
- * Mixed representations

Admissibility

Admissibility (\rightarrow) provided by pattern-matching functional programs

`zero` : nat

`succ` : nat \Rightarrow nat

double : nat \rightarrow nat

double `zero` = `zero`

double (`succ` n) = `succ` (`succ` (**double** n))

Admissibility

Can pattern-match on substitution functions as well:

eval : (exp \Rightarrow exp) \rightarrow rational \rightarrow rational

eval ($\lambda x.V$) r = **evalexp** (($\lambda x.V$) applied to (const r))


evalexp : exp \rightarrow rational

evalexp (const n) = n

evalexp (plus n1 n2) =

add (**evalexp** n1) (**evalexp** n2)

apply a substitution function
to perform substitution



Admissibility

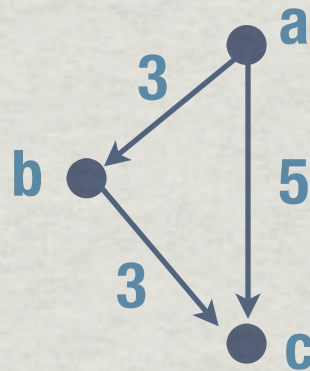
Twelf:

Can use **admissibility** to reason *about* logical systems, but not to *define* logical systems

Coq and our framework:

Can use **admissibility** in definitions

Admissibility



$\text{shortestPath}(a_1, a_2, n)$ iff $\text{path}(a_1, a_2, n)$ and
 $(\forall m. \text{path}(a_1, a_2, m) \rightarrow m \geq n)$

Admissibility

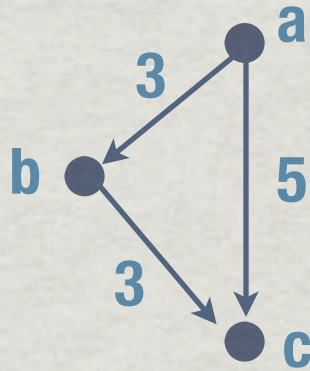
shortestPath(a_1, a_2, n) iff path(a_1, a_2, n) and
($\forall m. \text{path}(a_1, a_2, m) \rightarrow m \geq n$)

can be written as

path(a_1, a_2, n) ($\forall m. \text{path}(a_1, a_2, m) \rightarrow m \geq n$)

shortestPath(a_1, a_2, n)

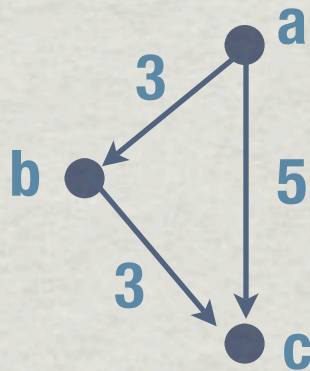
Admissibility



$\text{path}(a,c,5) \quad (\forall m.\text{path}(a,c,m) \rightarrow m \geq 5)$

$\text{shortestPath}(a,c,5)$

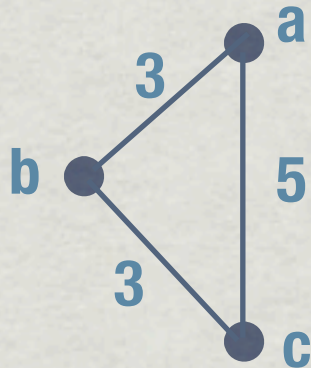
Admissibility in rules



path(a,c,5) case [ac]: $5 \geq 5$
 case [abc]: $6 \geq 5$

shortestPath(a,c,5)

Admissibility in rules



(undirected, repeats ok)

case [ac]: $5 \geq 5$

case [abc]: $6 \geq 5$

case [acbc]: $11 \geq 5$

path(a,c,5) ... infinitely many more ...

shortestPath(a,c,5)

Contrast with derivability

- * **Admissibility** ($\forall m.\text{path}(a,c,m) \rightarrow m \geq 5$) allows proof by cases on the paths in this particular graph
- * **Derivability** ($\forall m.\text{path}(a,c,m) \Rightarrow m \geq 5$) requires proof for a *new* path of *new* length m : **not true!**

Outline

- * Representations using **derivability**
- * Representations using **admissibility**
- * **Mixed representations**

Arithmetic expressions

Arithmetic expressions with let-binding:

```
let x be (const 4) in (plus x x)
```

```
e ::= const n  
    | let x be e1 in e2  
    | plus e1 e2  
    | times e1 e2  
    | sub e1 e2  
    | mod e1 e2  
    | div e1 e2
```

Arithmetic expressions

Arithmetic expressions with let-binding:

let x be (const 4) in (plus x x)

$e ::=$ const n
| let x be e1 in e2
| plus e1 e2
| times e1 e2
| sub e1 e2
| mod e1 e2
| div e1 e2

*Suppose we want
to treat binops
uniformly...*

Arithmetic expressions

Arithmetic expressions with let-binding

$e ::= \text{const } n$
| $\text{let } x \text{ be } e1 \text{ in } e2$
| $\text{binop } e1 \ \varphi \ e2$

where $\varphi : (\text{nat} \rightarrow \text{nat} \rightarrow \text{nat})$ is
the code for the binop.

Arithmetic expressions

`const` : `nat` \Rightarrow `exp`

`let` : `exp` \Rightarrow (`exp` \Rightarrow `exp`) \Rightarrow `exp`

`binop` : `exp` \Rightarrow (`nat` \rightarrow `nat` \rightarrow `nat`) \Rightarrow `exp` \Rightarrow `exp`

`let x be (const 4) in (x + x)`

represented by

`let (const 4) (λ x.binop x add x)`

where `add:(nat \rightarrow nat \rightarrow nat)` is the code for addition


Arithmetic expressions

`const` : `nat` \Rightarrow `exp`

`let` : `exp` \Rightarrow (`exp` \Rightarrow `exp`) \Rightarrow `exp`

`binop` : `exp` \Rightarrow (`nat` \rightarrow `nat` \rightarrow `nat`) \Rightarrow `exp` \Rightarrow `exp`

`let x be (const 4) in (x + x)`


`let (const 4) (λx.binop x add x)`

where `add`:(`nat` \rightarrow `nat` \rightarrow `nat`) is the code for addition

Arithmetic expressions

`const` : `nat` \Rightarrow `exp`

`let` : `exp` \Rightarrow (`exp` \Rightarrow `exp`) \Rightarrow `exp`

`binop` : `exp` \Rightarrow (`nat` \rightarrow `nat` \rightarrow `nat`) \Rightarrow `exp` \Rightarrow `exp`

`let x be (const 4) in (x + x)`

`let (const 4) (λ x. binop x add x)`

where `add`:(`nat` \rightarrow `nat` \rightarrow `nat`) is the code for addition

Arithmetic expressions

`const` : `nat` \Rightarrow `exp`

`let` : `exp` \Rightarrow (`exp` \Rightarrow `exp`) \Rightarrow `exp`

`binop` : `exp` \Rightarrow (`nat` \rightarrow `nat` \rightarrow `nat`) \Rightarrow `exp` \Rightarrow `exp`

`let x be (const 4) in (x + x)`

`let (const 4) (λx.binop x add x)`

where `add`:(`nat` \rightarrow `nat` \rightarrow `nat`) is the code for addition

Arithmetic expressions

`const` : `nat` \Rightarrow `exp`

`let` : `exp` \Rightarrow (`exp` \Rightarrow `exp`) \Rightarrow `exp`

`binop` : `exp` \Rightarrow (`nat` \rightarrow `nat` \rightarrow `nat`) \Rightarrow `exp` \Rightarrow `exp`

`let x be (const 4) in (x + x)`

`let (const 4) (λ x.binop x add x)`

where `add:(nat \rightarrow nat \rightarrow nat)` is the code for addition

Structural properties

Structural properties

Weakening for mixed representations:

Structural properties

Weakening for mixed representations:

1. Go from $(\text{nat} \Rightarrow \text{nat})$ to $(\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat})$

Structural properties

Weakening for mixed representations:

1. Go from $(\text{nat} \Rightarrow \text{nat})$ to $(\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat})$

$\lambda x.(\lambda y.V)$ [x doesn't occur in V]

Structural properties

Weakening for mixed representations:

1. Go from $(\text{nat} \Rightarrow \text{nat})$ to $(\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat})$

$\lambda x. (\lambda y. V)$ [x doesn't occur in V]

2. Go from $(\text{nat} \rightarrow \text{nat})$ to $(\text{nat} \rightarrow \text{nat} \rightarrow \text{nat})$

Structural properties

Weakening for mixed representations:

1. Go from $(\text{nat} \Rightarrow \text{nat})$ to $(\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat})$

$\lambda x.(\lambda y.V)$ [x doesn't occur in V]

2. Go from $(\text{nat} \rightarrow \text{nat})$ to $(\text{nat} \rightarrow \text{nat} \rightarrow \text{nat})$

$\text{drop } x \ y = f \ y$ [x doesn't occur in f]

Structural properties

Structural properties

Weakening for mixed representations:

Structural properties

Weakening for mixed representations:

3. Go from $(\text{nat} \Rightarrow \text{nat})$ to $(\text{nat} \rightarrow \text{nat} \Rightarrow \text{nat})$

Structural properties

Weakening for mixed representations:

3. Go from $(\text{nat} \Rightarrow \text{nat})$ to $(\text{nat} \rightarrow \text{nat} \Rightarrow \text{nat})$

$h\ x = \lambda y.V$ [x doesn't occur in V]

Structural properties

Weakening for mixed representations:

3. Go from $(\text{nat} \Rightarrow \text{nat})$ to $(\text{nat} \rightarrow \text{nat} \Rightarrow \text{nat})$

$h\ x = \lambda y.V$ [x doesn't occur in V]

4. Go from $(\text{nat} \rightarrow \text{nat})$ to $(\text{nat} \Rightarrow (\text{nat} \rightarrow \text{nat}))$

Structural properties

A function of type $(\text{nat} \rightarrow \text{nat})$:

`double zero = zero`

`double (succ n) = succ (succ (double n))`

Structural properties

A (nat \Rightarrow (nat \rightarrow nat)):

$\lambda x.$ double zero = zero

double (succ n) = succ (succ (double n))

double x = ???

*Problem: double doesn't have a case for x,
so mixed representations not necessarily structural!*

Structural properties

Our solution:

- * $\lambda x.V$ used by pattern-matching; application (=substitution) is not built-in
- * Nothing forces \Rightarrow to be structural
- * Weakening/substitution implemented generically for a wide class of rule systems, using some technical conditions

Structural properties

`const` : $\text{nat} \Rightarrow \text{exp}$

`let` : $\text{exp} \Rightarrow (\text{exp} \Rightarrow \text{exp}) \Rightarrow \text{exp}$

`binop` : $\text{exp} \Rightarrow (\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \Rightarrow \text{exp} \Rightarrow \text{exp}$

- * Can't weaken `exp` with `nat`:
could need new case for \rightarrow in a `binop`
- * Can weaken `exp` with `exp`:
doesn't appear to left of \rightarrow

Conclusion

Summary

- * **Derivability (variable binding)** provided by substitution functions (\Rightarrow)
- * **Admissibility (arbitrary reasoning)** provided by pattern-matching functional programs (\rightarrow)
- * Our framework makes it easy to use both

Future work

- * This talk: syntax of logical systems
Need **dependent types** to represent judgements
- * More examples mixing \Rightarrow / \rightarrow
- * Thesis work: a language in which you can define logics and use them to reason about programs

Strange Case of
Dr. Admissibility
and
Mr. Derive